Efficient State Management in Distributed Ledgers

Dimitris Karakostas
Nikos Karayiannidis
Aggelos Kiayias

Types of shared state

1. *Public ledger:* the ordered list of published transactions

2. *Mempool:* the set of unpublished transactions

3. **Active state**:
    a. UTxO set
    b. Global state (Ethereum)

Attacks against active state

The Ethereum network is currently undergoing a DoS attack

Posted by Jeffrey Wilcke on September 2016

Research & Development

July 2015 flood attack

The **July 2015 flood attack** was a large "stress test" of the Bitcoin network. The possib
subsequent to stress tests executed in June.

Overview

- UTxO-based ledger model
- Transaction-oriented state optimization
  - Rule based (heuristic)
  - Transformations
  - Exhaustive search of transaction space
- Fee model
  - The *state efficiency* property
  - A state efficient Bitcoin

UTxO Ledger

- $\mathcal{L} \stackrel{def}{=} List[Transaction]$
- $Transaction \stackrel{def}{=} (inputs : Set[Input],$
  $outputs : List[Output],$
  $forge : Value,$
  $fee : Value)$
- $UTxO \stackrel{def}{=} (\alpha : Address,\ value : Value,\ created : Timestamp)$
- $Input \stackrel{def}{=} (id : Hash, index : Int)$
- $State \stackrel{def}{=} Set[Input]$
- $\mathsf{costTx} : UtxoTx -> LedgerState -> Cost$
  $\mathsf{costTx}(\tau, \mathcal{S}) = cost(\mathsf{txRun}(\tau, \mathcal{S})) - cost(\mathcal{S})$

Transaction Model

- Transaction equivalency:

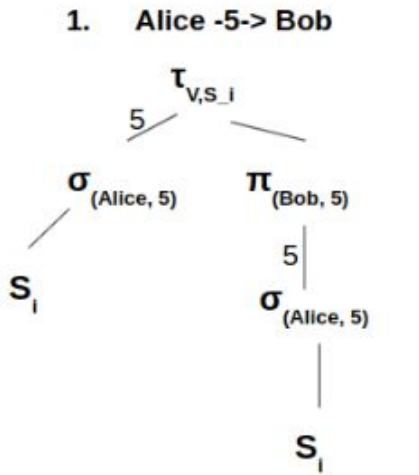
$$\forall \alpha \in A \quad \sum_{\substack{i \in \mathcal{S}_1 \\ o_i = out(i,\mathcal{L}) \\ o_i.address = \alpha}} o_i.value = \sum_{\substack{j \in \mathcal{S}_2 \\ o_j = out(j,\mathcal{L}) \\ o_j.address = \alpha}} o_j.value$$

where A is the set of all addresses of the parties participating in the distributed ledger system.

- **Input Selection** $\sigma_{[(a_1,v_1),\ldots,(a_n,v_n)]} : LedgerState \rightarrow LedgerState$
- **Output Creation** $\pi_{[(a_1,v_1),\ldots,(a_n,v_n)]} : LedgerState \rightarrow LedgerState$
- **Transaction Validation** $\tau_{V,\mathcal{S}_i} : LedgerState \rightarrow LedgerState \rightarrow LedgerState$

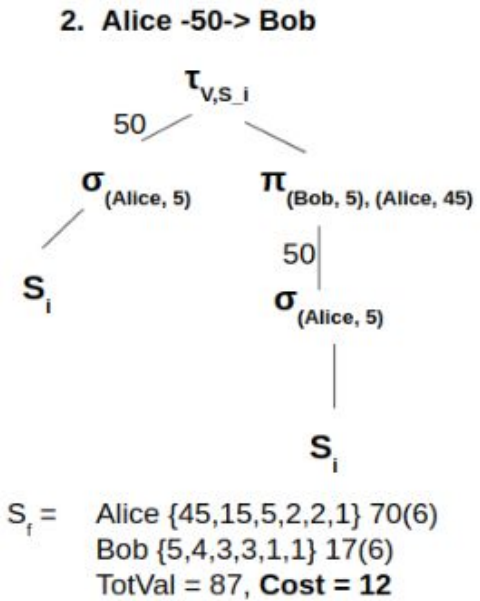
Alice wants to give Bob 5 coins

- Alice: {50, 15, 5, 2, 2, 1} (75 total coins, 6 UTxOs)
- Bob: {4, 3, 3, 1, 1} (12 total coins, 5 UTxOs)



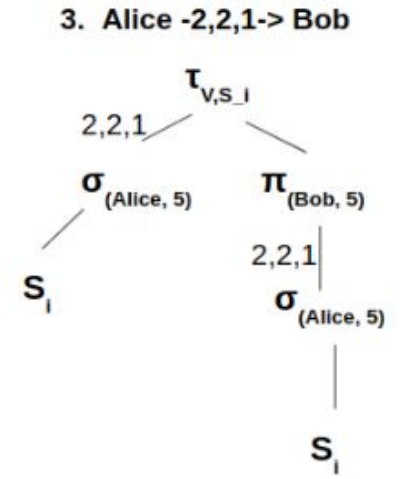
Alice wants to give Bob 5 coins

- Alice: {50, 15, 5, 2, 2, 1} (75 total coins, 6 UTxOs)
- Bob: {4, 3, 3, 1, 1} (12 total coins, 5 UTxOs)



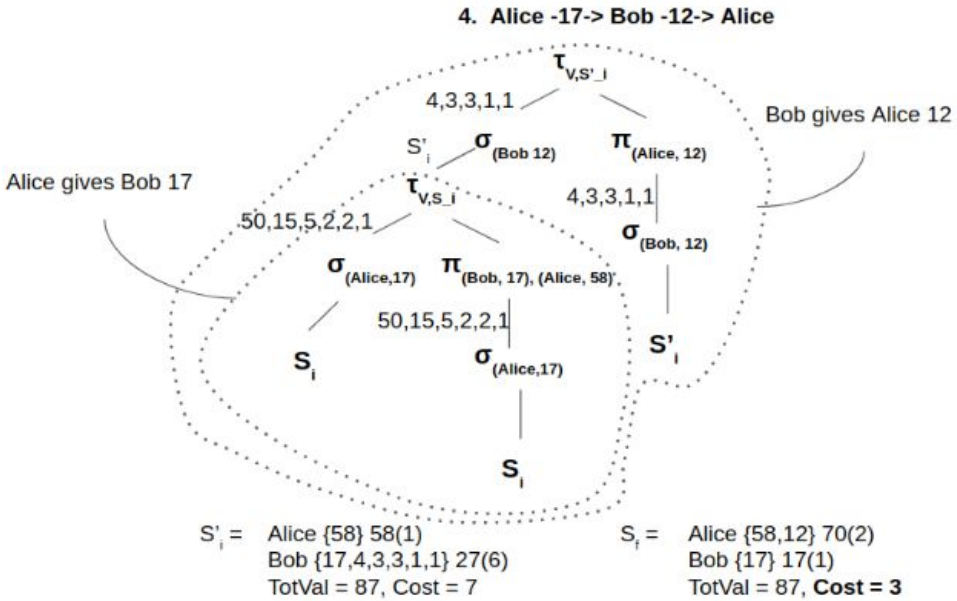
Alice wants to give Bob 5 coins

- Alice: {50, 15, 5, 2, 2, 1} (75 total coins, 6 UTxOs)
- Bob: {4, 3, 3, 1, 1} (12 total coins, 5 UTxOs)



Alice wants to give Bob 5 coins

- Alice: {50, 15, 5, 2, 2, 1} (75 total coins, 6 UTxOs)
- Bob: {4, 3, 3, 1, 1} (12 total coins, 5 UTxOs)



Transaction Optimization

- Rules
  - One output per address
  - Consume maximum # of inputs
  - …
- Transformations
  - Reorder tx creation to minimize cost
  - …
- Methods
  - Input selection algorithm
  - ...
- Search of transaction space
  - Optimization metrics (eg. cost, size)
  - Limits (eg. max fees to spend)

State-aware input selection

- *Problem*: select inputs to consume
- *Goal*: minimize state cost + upper cap on fees
- *Fee model*:
  - input & output sizes fixed
  - fees = (numOf(inputs) + numOf(outputs) + 1) * utxoSizeBytes * costPerByte
- *State-aware input selection algorithm*:
  - Sort inputs per value
  - Find largest window such that:
    - total value of inputs ≥ tx output
    - fees of consuming inputs ≤ upper cap

State Efficient Fee Model

State Efficiency:

$$\forall \mathcal{S} \in \mathbb{S} \forall \tau_1, \tau_2 \in \mathbb{T} \mid \tau_1 \equiv \tau_2 \wedge costTx(\tau_1, \mathcal{S}) > costTx(\tau_2, \mathcal{S}) : F(\tau_1, \mathcal{S}) > F(\tau_2, \mathcal{S})$$

Bitcoin

- F(tx) = sizeOf(tx) * feePerByte
- Simple model:
  - i: size of consuming an input
  - o: size of UTxO
- Example:
  - tx_1: 1 input, 1 output
    - $cost(tx_1) = 0$
    - $F(tx_1) = (i + o) * feePerByte$
  - tx_2: 2 inputs, 1 output
    - $cost(tx_2) = 1$
    - $F(tx_2) = (2i + o) * feePerByte = F(tx_1) + i*feePerByte$

State-efficient Bitcoin

- F’(tx) = sizeOf(tx) * feePerByte + cost(tx) * feePerUTxO
- Intuition:
  - When a user creates a UTxO (increasing the state), they pay an extra fee
  - When a user consumes an input (decreasing the state), they get reimbursed
- If *feePerUTxO > i * feePerByte*, fee function is state efficient
  - The user who creates a UTxO reimburses the (future) user who consumes it for at least enough fees to cover the increase in tx size

Ethereum

- selfdestruct:
  - destroys the smart contract
  - frees up global state
- Programmer who writes the *selfdestruct* line of code pays fee for it (due to increased smart contract size)
- Gives user who *runs* it with (gas) fee discount
- User who receives discount ≠ User who pays the fee
- Discount is *at most* half transaction's fees
  - User has no incentive to *only* run *selfdestruct*

Summary

- Poor state management may lead to:
  - Bad performance, i.e. bloated state shared across all parties
  - Denial-of-Service
- Transaction optimization should take into account state efficiency
- Fees should reflect state cost of transaction
- Paper’s full version available: https://eprint.iacr.org/2020/525.pdf

Thank you!